COURSE CODE: CMP 112
COURSE TITLE: PROGRAMMING ESSENTIALS
NUMBER OF UNITS: 3 Units
COURSE DURATION: Three hours per week

COURSE LECTURER: **ACHEME DAVID**

INTENDED LEARNING OUTCOMES

At the completion of this course, students are expected to:

1. Define the concept of programming and levels of programming language
2. Understand all the programming paradigms
3. Understand the concept of algorithms and methods of representing algorithms
4. Apply the concept of algorithm to express solutions to simple problems
5. Write simple programs in QBasic Programming language.

COURSE DETAILS:

Week 1-2: Introduction to concept of programming logic, programs, Levels of programming languages

Week 3-4: Concept of algorithms: flowcharts and pseudocode with emphasis on; Develop algorithms to solve a wide range of common programming problems

Week 5-6: Introduction to the QBASIC Programming Language, the syntax data types and concept of variables

Week 7-8: Design, implement, debug and test small programs using at Qbasic Programming language

Week 9-10: Use common programming tools such as compilers, editors and debuggers.

Week 11: Compare and contrast the different paradigms, understanding the relative advantages and disadvantages of each; (d)

Week 12: Revision

RESOURCES

• **Lecturer's Office Hours:**
• Dr. Japheth B. R. Mondays 12:30-2:30pm.
• Mr. Acheme David, Wednesdays 2-4pm,
• **Course lecture Notes: http://www.edouniversity.edu.ng/oer/compsc/cmp122.pdf**
• **Books:**
• *Practical Introduction to Data Structures and Algorithm Analysis*, C++ Edition, 2nd
Edition by Clifford A. Shaffer. Prentice Hall, 2000. ISBN: 0-13-028-446-7.
• *Foundations of Multidimensional and Metric Data Structures* by Hanan Samet. Morgan Kaufmann, 2006. ISBN: 0-12-369-446-9 (recommended).

Programming Project:
• Multiple parts (2 or 3).
• Must be done in C/C++
• Homework + Project: ~ 30% of final grade.
• **Exams:**
• Final, comprehensive (according to university schedule): ~ 70% of final grade

Assignments & Grading

• **Academic Honesty:** All classwork should be done independently, unless explicitly stated otherwise on the assignment handout.
• You may discuss general solution strategies, but must write up the solutions yourself.
• If you discuss any problem with anyone else, you must write their name at the top of your assignment, labeling them "collaborators".
• **NO LATE HOMEWORKS ACCEPTED**
• Turn in what you have at the time it's due.
• All homeworks are due at the start of class.
• If you will be away, turn in the homework early.
• Late Programming Assignments (projects) will not be accepted, but penalized according to the percentages given on the syllabus.

PREAMBLE:

Computers can perform different tasks. In school, students use computers for tasks such as writing papers, searching for articles, sending email, and participating in online classes. At work, people use computers to analyze data, make presentations, conduct business transactions, communicate with customers and coworkers, control

machines in manufacturing facilities, and do many other things. At home, people use computers for tasks such as paying bills, shopping online, communicating with friends and family, and playing computer games. And don't forget that cell phones, iPods, BlackBerries, car navigation systems, and many other devices are computers too. The uses of computers are almost limitless in our everyday lives.

Computers can do such a wide variety of things because they are programmed to do so. This means that computers are not designed to do just one job, but to do any job that their programs tell them to do. By definition, ***a computer system is a device that accepts input known as data from the user, process the inputs based on a set of instructions called program and produce the results or output also called information***.

A *program* is a set of instructions that a computer follows to perform a task, for example, Microsoft Word and Adobe Photoshop. Microsoft Word is a word processing program that allows you to create, edit, and print documents with your computer. Adobe Photoshop is an image editing program that allows you to work with graphic images, such as photos taken with your digital camera.

These Programs are commonly referred to as *software*. Software is essential to a computer because it controls everything the computer does. All of the software that we use to make our computers useful is created by individuals working as programmers or software developers. A *programmer*, or *software developer*, is a person with the training and skills necessary to design, create, and test computer programs. Computer programming is an exciting and rewarding career. Today, you will find programmers' work used in business, medicine, government, law enforcement, agriculture, academics, entertainment, and many other fields.

This course introduces you to the fundamental concepts of computer programming using the beginners' language known as BASIC programming language.

OVERVIEW OFPROGRAMMING

Computer programming is a problem-solving approach. Basically, programmers develop programs when encountered with challenges of proffering solutions to real life problems.A computer program has immense potential for saving time/energy, as most computational tasks that are repetitive or can be generalized can be done by

computer programs. Programs can be built using programming languages which are classified into three categories viz;

a) Machine language: This is the only language the computer understands and is also called a binary language. It is made up of 0's and 1's.
b) Low Level languages: These are assembly languages that use mnemonics to represents computer operations. The mnemonics are translated into machine language using a translator program known as the assembler. An Assembly language is this symbolic language used to enter machine code instructions using easy-to-remember mnemonics. The assembler converts assembly language statements into machine codes.
c) High Level Languages (HLL): These are computer languages that allow the programmer or software developer to develop program codes using human understandable expressions. HLL uses an intermediate translator program known as **Compiler** or **interpreter** to converts program codes written in HLL to machine understandable 0's and 1's for the computer to execute the programs. Examples include; BASIC, Java, C++, VB.Net, python, Pascal, FORTRAN, PHP, etc.

This course intends to introduce students to computer programming using high level programming languages. We shall adopt the beginners programming language known as BASIC (**Beginner's All-Purpose Symbolic Instruction Code**) to get students acquainted with computer programs. The Basic programming language is interpreted and therefore requires an **interpreter** program in order to run any program written in the Basic programming language.

ALGORITHMS

Recall that programming is a problem-solving approach. To develop programs that will solve human problems efficiently, there is a need for a step-by-step representation of the solutions before transforming them into computer programs using specific computer languages. An algorithm is a representation of a solution to a problem. It is procedure for solving a problem in finite number ofsteps. Algorithms provide step-by-step methods of accomplishing a task.

The term **algorithm** originally referred to any computation performed via a set of rules applied to numbers written in decimal form. The word is derived from the

phonetic pronunciation of the last name of *Abu Ja'far Mohammed ibn Musa al-Khowarizmi*, who was an Arabic mathematician who invented a set of rules for performing the four basic arithmetic operations (addition, subtraction, multiplication and division) on decimal numbers.

*An **algorithm** is procedure consisting of a finite set of unambiguous rules (instructions) which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems for any allowable set of input quantities (if there are inputs). In other word, an **algorithm** is a step-by-step procedure to solve a given problem*
Alternatively, we can define an algorithm as a set or list of instructions for carrying out some process step by step.

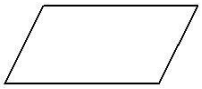
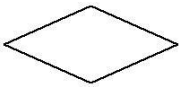
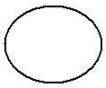
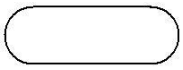
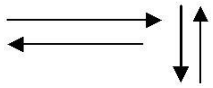
In the problem-solving phase of computer programming, you will be designing algorithms. This means that you will have to be conscious of the strategies you use to solve problems in order to apply them to programming problems. These algorithms can be designed or implemented using **flowcharts** or **pseudocode**.

FLOWCHARTS

Flowcharting is a tool developed in the computer industry, for showing the stepsinvolved in a process. A flowchart is a diagram made up of *boxes*, *diamonds* and other shapes, *connected by arrows* - each shape represents a step in the process, and the arrows show the order in which they occur. Flowcharting combines symbols and flow lines, to show figuratively the operation of an algorithm.

Flowcharting Symbols

There are 6 basic symbols commonly used in flowchating: Terminal, Process, input/output, Decision, Connector and Predefined Process. This is not a complete list of all the possible flowcharting symbols; it is the list of ones used most often in structured programming.

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the Processor or Memory
	input/output	Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results
	Decision	Used to ask a question that can be answered in a binary format (Yes/No, True/False)
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow.
	Predefined Process	Used to invoke a subroutine or an interrupt program.
	Terminal	Indicates the starting or ending of the program, process, or interrupt program.
	Flow Lines	Shows direction of flow.

Generally, there are many standard flowcharting symbols.

General Rules for flowcharting

1. All boxes of the flowchart are connected with Arrows. (Not lines)
2. Flowchart symbols have an entry point on the top of the symbol with no other entry points. The exit point for all flowchart symbols is on the bottom except for the Decision symbol.
3. The Decision symbol has two exit points; these can be on the sides or the bottom and one side.
4. Generally a flowchart will flow from top to bottom. However, an upward flow can be shown as long as it does not exceed 3 symbols.
5. Connectors are used to connect breaks in the flowchart. Examples are:
    - From one page to another page.
    - From the bottom of the page to the top of the same page.
    - An upward flow of more than 3 symbols
6. Subroutines and Interrupt programs have their own and independent

flowcharts.

7. All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.
8. All flowcharts end with a terminal or a contentious loop.

Flowcharting uses symbols that have been in use for a number of years to represent the type of operations and/or processes being performed. The standardized format provides a common method for people to visualize problems together in the same manner. The use of standardized symbols makes the flow charts easier to interpret; however, standardizing symbols is not as important as the sequence of activities that make up the process.

Examples of Algorithms and Flowcharts

Example 1: Design an algorithm and the corresponding flowchart for adding the testscores as given below:

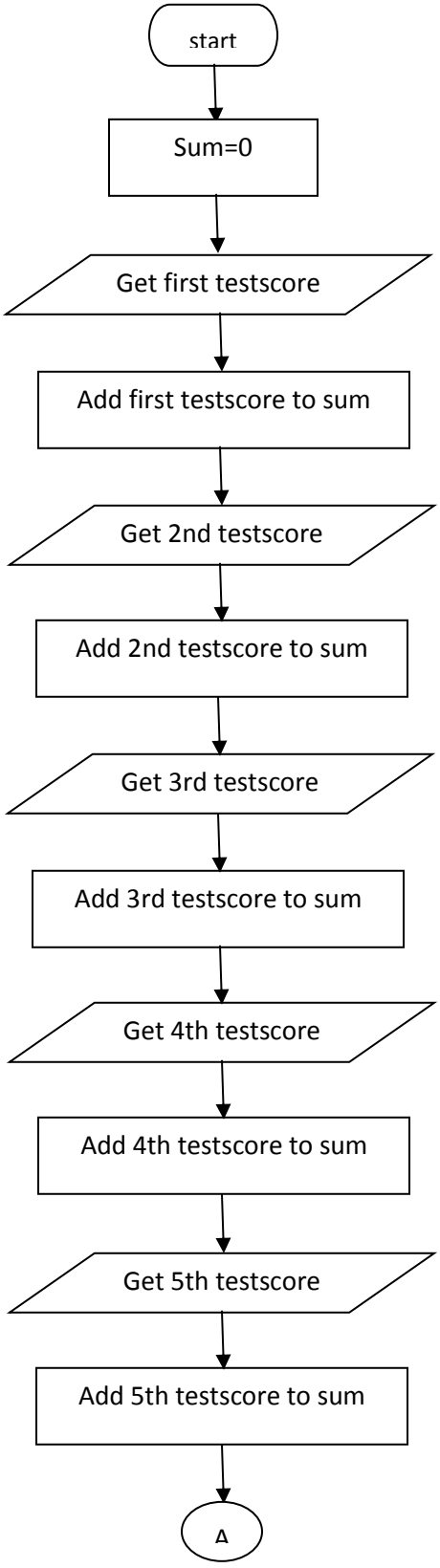
26, 49, 98, 87, 62, 75

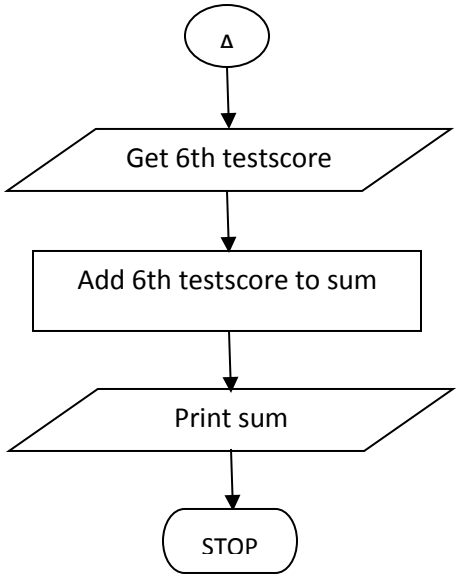
a) Algorithm

1. Start
2. Sum = 0
3. Get the first testscore
4. Add first testscore to sum
5. Get the second testscore
6. Add to sum
7. Get the third testscore
8. Add to sum
9. Get the Forth testscore
10. Add to sum
11. Get the fifth testscore
12. Add to sum
13. Get the sixth testscore
14. Add to sum
15. Output the sum
16. Stop

b) The corresponding flowchart is as follows:

c)

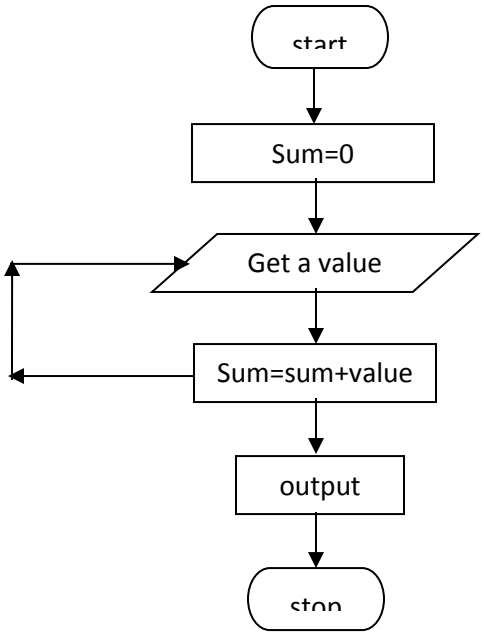

start
Sum=0
Get first testscore
Add first testscore to sum
Get 2nd testscore
Add 2nd testscore to sum
Get 3rd testscore
Add 3rd testscore to sum
Get 4th testscore
Add 4th testscore to sum
Get 5th testscore
Add 5th testscore to sum
A



The algorithm and the flowchart above illustrate the steps for solving the problem of adding six test scores. Where one test score is added to sum at a time. Both the algorithm and flowchart should always have a **Start** step at the beginning of the algorithm or flowchart and at least one **stop** step at the end, or anywhere in the algorithm or flowchart. Since we want the sum of six test scores, then we should have a container for the resulting sum. In this example, the container is called **sum** and we make sure that sum should start with a zero value by step 2.

Example 2: The problem with this algorithm is that, some of the steps appear more thanonce, i.e. step 5 get second number, step 7, get third number, etc. One could shorten the algorithm or flowchart as follows:

1. Start
2. Sum = 0
3. Get a value
4. sum = sum + value
5. Go to step 3 to get next Value
6. Output the sum
7. Stop

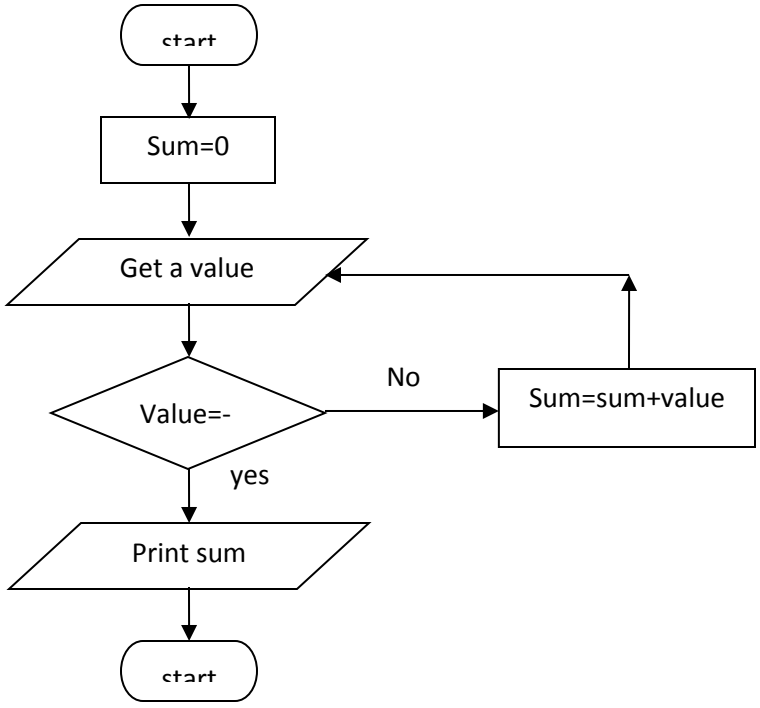


This algorithm and its corresponding flowchart are a bit shorter than the first one. In this algorithm, step 3 to 5 will be repeated, where a number is obtained and added to sum. Similarly the flowchart indicates a flow line being drawn back to the previous step indicating that the portion of the flowchart is being repeated. One problem indicates that these steps will be repeated endlessly, resulting in an **endless** algorithm or flowchart. The algorithm needs to be improved to eliminate this problem. In order to solve this problem, we need to add a last value to the list of numbers given. This value should be unique so that, each time we get a value, we test the value to see if we have reached the last value. In this way our algorithm will be a finite algorithm which ends in a finite number of steps as shown below. There are many ways of making the algorithm finite.

The new list of numbers will be 26, 49, 498, 9387, 48962, 1, -1. The value –1 is a unique number since all other numbers are positive.

1. Start
2. Sum = 0
3. Get a value
4. If the value is equal to –1, go to step 7
5. Add to sum (sum = sum + value)
6. Go to step 3 to get next Value
7. Output the sum
8. Stop

Corresponding flowchart:



PSEUDOCODE

Pseudocode is one of the tools that can be used to write a preliminary plan that can bedeveloped into a computer program. **Pseudocode** is a generic way of describing an algorithm without use of any specific programming language syntax. It is, as the name suggests, *pseudo* code —it cannot be executed on a real computer, but it models and resembles real programming code, and is written at roughly the same level of detail.

Pseudocode, by nature, exists in various forms, although most borrow syntax from popular programming languages (like **C**, **Lisp**, or **FORTRAN**). Natural language is used whenever details are unimportant or distracting.

In the algorithm design, the steps of the algorithm are written in free English text and, although brevity is desired, they may be as long as needed to describe the particular operation. The steps of an algorithm are said to be written in pseudocode.

Many languages, such as Pascal, have a syntax that is almost identical to pseudocode and hence make the transition from design to coding extremely easy. Examples of pseudocodes include; suppose you are required to design an algorithm for finding the average of six numbers and the sum of the numbers is given. The pseudocode

will be as follows:

Start
Get the sum
Average = sum / 6
Output the average
Stop

This is the pseudo-code required to input three numbers from the keyboardand output the result.

Use variables: sum, number1, number2, number3 of type integer
Accept number1, number2, number3
Sum = number1 + number2 + number3
Print sum
End program

The following pseudo-code describes an algorithm which will accept twonumbers from the keyboard and calculate the sum and product displaying the answer on the monitor screen.

Use variables sum, product, number1, number2 of type
real display "Input two numbers"
accept number1, number2 sum
= number1 + number2 print
"The sum is ", sum
*product = number1 * number2*
print "The Product is ", product
end program

The program is to input a examination mark and test it for the award of agrade. The mark is a whole number between 1 and 100. Grades are awarded according to the following criteria:

>= 80 Distinction
>= 60 Merit
>= 40 Pass
< 40 fail

The pseudo-code is

Use variables: mark of type integer
If mark >= 80 display "distinction"

If mark >= 60 and mark < 80 display "merit"
If mark >= 40 and mark < 60 display "pass"
If mark < 40 display "fail"

PROGRAMMING PARADIGMS

WHAT IT IS:

The word paradigm is used a great deal when talking about programming languages. What does it mean?

A *programming paradigm* is a style or "way" of programming. Some languages make it easy to write in some paradigms but not others.

Some of the more common paradigms are

- **Imperative** — Control flow is an explicit sequence of commands.
- **Declarative** — Programs state the result you want, not how to get it.
- **Structured** — Programs have clean, goto-free, nested control structures.
- **Procedural** — Imperative programming with procedure calls.
- **Functional (Applicative)** — Computation proceeds by (nested) function calls that avoid any global state.
- **Function-Level (Combinator)** — Programs have no variables. No kidding.
- **Object-Oriented** — Computation is effected by sending messages to objects; objects have state and behavior.
    - **Class-based** — Objects get their state and behavior based on membership in a class.
    - **Prototype-based** — Objects get their behavior from a prototype object.
- **Event-Driven** — Control flow is determined by asynchronous actions (from humans or sensors).
- **Flow-Driven** — Computation is specified by multiple processes communicating over predefined channels.
- **Logic (Rule-based)** — Programmer specifies a set of facts and rules, and an engine infers the answers to questions.
- **Constraint** — Programmer specifies a set of constraints, and an engine infers the answers to questions.
- **Aspect-Oriented** — Programs have cross-cutting concerns applied transparently.

- **Reflective** — Programs manipulate their own structures.
- **Array** — Operators are extended to arrays, so loops are normally unnecessary.

Paradigms are **not meant to be mutually exclusive**; you can program in a functional, object-oriented, event-driven style.

TWO POPULAR PROGRAMMING PARADIGMS

Object-Oriented Programming Paradigm

OOP is based on the sending of messages to objects. Objects respond to messages by performing operations. Messages can have arguments, so "sending messages" looks a lot like calling subroutines. A society of objects, each with their own "local memory" and own set of operations has a different feel than the "monolithic processor and single shared memory" feel of non object oriented languages.

The first object oriented language was Simula-67; Smalltalk followed soon after as the first "pure" object-oriented language. Many languages designed from the 1980s to the present have been object-oriented, notably C++, CLOS (object system of Common Lisp), Eiffel, Modula-3, Ada 95, Java, C#, Ruby.

Structured Programming Paradigm

Programs have clean, goto-free, nested control structures. Structured programming is a kind of imperative programming where the control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos. Variables are generally local to blocks (have lexical scope).

INTRODUCTION TO QBASIC

BASIC stands for Beginner's All Purpose Symbolic Instruction Code. It was invented in 1963, at Dartmouth College, by the mathematicians John George Kemeny and Tom Kurtzas.

BASIC is an interpreter which means it reads every line, translates it and lets the computer execute it before reading another. Each instruction starts with a line number.

FEATURES OF QBASIC

1. It is a user friendly language.
2. It is widely known and accepted programming language.
3. It is one of the most flexible languages, as modification can easily be done in already existing program.
4. Language is easy since the variables can be named easily and uses simple English phrases with mathematical expressions.

RULES OF QBASIC

Every programming language has a set of rules that have to be followed while writing a program; following are some rules of QBASIC language:

1. All QBasic programs are made up of series of statements, which are executed in the order in which they are written.
2. Every statement should have at least one QBasic command word. The words that BASIC recognizes are called keywords.
3. All the command words have to be written using some standard rules, which are called "Syntax Rules". Syntax is the grammar of writing the statement in a language. Syntax Errors are generated when improper syntax is detected.
4. Basic program files must be saved with a file extension name of **.bas**. For a QBASIC program to run, it must be saved.

DATA TYPES

Data is a collection of facts and figures that is entered into the computer through the keyboard. Data is of two types:

1. CONSTANT: Data whose value does not change or remains fixed. A constant is a data object whose value cannot be changed. There are two types of constants:

(a) NUMERIC CONSTANT: Numbers -negative or positive used for mathematical calculations e.g. –10, 20, 0

(b) ALPHANUMERIC CONSTANT / STRING: Numbers or alphabets written within double quotes (inverted commas " "). e.g. "Computer", "Operating System"
2.VARIABLE: A ***variable*** is a data object whose value can be defined and redefined. A variable can simply be referred to as a name which can contain a value. It is a data whose value is not constant and may change due to some calculation during the program execution. A variable is actually a location in the computer's memory, which stores the values. Depending on what value is held, Variables are of two types:

- NUMERIC VARIABLE: The variable that holds a Numeric data values for arithmetic calculations (+, - ,*, /) is called a Numeric Variable. e.g. A = 50, here A is the Numeric Variable. In numeric variables, the symbol % means integer and it truncate real numbers eg: A% = 6.2 implies A%=6
  A%=6.6 implies A% = 7 etc. the symbols !, & and # represents single, long and double precision variables respectively.
- CHARACTER OR STRING VARIABLE: The variable that holds an Alphabetic, Alphanumeric or numeric data value, which cannot be used for arithmetic calculations, is called Character Variable or String Variable. This variable must end with a $ sign and the value it hold must be enclosed in inverted commas. e.g. Name$ = "BringBackOurGirls", here Name$ is a string Variable. If a variable holds a symbol or text, it may be a character variable or a string variable. E.g; C$, ICT101$, CE$, etc.

Both constant names and variable names are user defined and must not start with a special symbol or numeric values. QBASIC doesn't allow keywords to be used as variable or constant names. Variables and Constant names utilizes a character set of A-Z and a-z alphabetic values,0-9 numeric values and a special dot (.) symbol might be used.

ASSIGNMENT STATEMENTS

In programming, *assignment* is the term for setting a variable equal to some value. Assignment allocates data values to the memory locations represented by the variables on the left hand side of the said statement. These statements assign an expression to a quantity. The general form is:

Let <variable> = <value/expression>

Where **expression** may be a variable, a numeric dataand or an arithmetic expression, or combination of each.

An assignment statement uses the equals sign (=), but this symbol does not have the same meaning as it has in mathematics. In QBasic, the statement

x = x + 1

means that the value x is to be incremented by 1 and the result stored into the memory location reserved for the quantity represented by x. Examples of assignment statements include;
Let A = 15…….. Assigning a numeric value to a variable
Let A = B……Assigning variable to a variable
Let C = A + B…. Assigning an expression to a variable

In QBASIC, assignment statements can also be written without including the LET keyword: **<variable> = <value/expression>**
The keyword LET is optional in an assignment statement. Examples include;
A = 15…….. Assigning a numeric value to a variable
A = B……Assigning variable to a variable
C = A + B…. Assigning an expression to a variable

OPERATORS
There are different types of operators in FORTRAN 95, the common ones include: arithmetic, relational and logical.
ARITHMETIC OPERATORS: They include;

Operator	**Function**	**Example**	**Results**
+	Addition	10+5	15
-	Subtraction	10-5	5
*	Multiplication	10*5	50
/	Division	10/5	2
^	Exponentiation	10^5	100000

RELATIONAL OPERATORS: Relational operators are operators that are placed between expressions and that compare the results of the expressions. They are a set of binary *relational operators* that take any two numeric or character variables of the same type and kind and return a logical result. These operators are used in *conditionals* to construct tests. The relational operators in QBASIC are:

operator	meaning	example
=	Equal to	IF a = 15 THEN…
<>	Not equal to	IF a <> 15 THEN…
<	Less than	IF a < 15 THEN…
<=	Less or equal to	IF a <= 15 THEN
>	greater than	IF a > 15 THEN…
>=	greater or equal to	IF a >= 15 THEN…

LOGICAL OPERATORS

The type logical can have only two different values: true and false. These are variables that can only take Boolean values. Logical variables can be operated upon by logical operators. These are:

OPERATOR	MEANING
OR	Logical OR
AND	Logical AND
NOT	Compliment/Negation

QBASIC EXPRESSIONS AND STATEMENTS

An **expression** is a combination of symbols that forms a valid unit for evaluation or computation. Examples include

$$x = \frac{a^3 - 7b^2}{3b + \sqrt{d}}$$

The mathematical expression above can be express in QBASIC assignment statement as follow;

X = (a^3 – 7*b^2)/(3*b + d^(1/2))

INPUT/OUTPUT STATEMENTS

Data can be transferred into the program using input statement and the results be communicated to the user or the outside world using the output statement. In

programming, these statements are generally known as **input/output** statements or simply **i/o statements**.

INPUT: This statement allows the user to enter a value for the variable while running the program. A question mark (?) appears on the output screen waiting for the user to enter a relevant data and then press enter key. Once the Return key or Enter key is pressed the data is stored in the variable.
SYNTAX : **INPUT < VARIABLE >**
Examples include;
Input A……Enter Numeric constant
Input "Enter name:";N$….Giving relevant message to avoid erroneous data input

OUTPUT STATEMENTS: This statement displays the results or the required output of a QBASIC program on the screen for the user to visualize. It is implemented using the PRINT statement with the syntax as shown below;

PRINT < VARIABLE >

Examples include;

a) PRINT "My name is Nick."
   Type the bolded text into QBasic and press F5 to run the program. On the screen you'll see: My name is Nick.
   Note: you must put the text in quotes, like this – "text". The text in quotes is called a string. If you put the PRINT alone, without any text, it will just put an empty line. PRINT can also put numbers on the screen.
b) PRINT 57 will show the number 57. This command is useful for displaying the result of mathematical calculations. But for calculations, as well as for other things in the program, you need to use variables.
c) Given an assignment statement; a = 35; the print statement below can output the value of a on the screen:
   PRINT a
d) Suppose a$= "welcome to the computing world", PRINT a$ can display **welcome to the computing world** on the computer screen
e) The PRINT command can print more than one string on the line. To do this, put the ; sign between the variables. For example, you have two variables – name$ and age, where name$ contains Doowuese, and age contains the number 12. Then, to print both name and age, you type:
   PRINT "Her name is"; name$; ". She's age: "; age
   What you see on the screen when you run the program will look like this:

Her name is Doowuese. She's age: 12

Or, you can type a program statement like this:

PRINT "Her name is :"; name$
PRINT "She's age"; age
The result is:
Her name is Doowuese
She's age 12

f) Print with Semi-Colon (;) Semi-colon placed after the message to be displayed, leaves no space between two messages.
E.g. Print "This is an example";" of QBasic program"
output: This is an example of QBasic program

g) Print with Comma(,): The screen of the computer is made of 80 columns and 40 rows. The columns are divided into five (5) zones of 14 columns each. Comma placed after the message prints the message zone wise on the screen.

THE REM STATEMENT

REM: It stands for Remark. It gives an explanation of the program or of the statements in the program thereby making the program more understandable to the reader. It is used for making comments and hence the computer does not execute this statement and whatever is written after REM is ignored by the interpreter. REM can be used anywhere and many times in a program. E.g.
REM : this program introduces chemical engineering students to QBASIC programs

THE END STATEMENT

END: This command is usually given at the end of the program. Statements written after end are not executed since the program terminates execution on reading this command.

THE CLS STATEMENT

CLS: This command is used to clear the screen. It is written at the beginning of a program.

THE DELETE STATEMENT

DELETE <LINE NO.>: To delete a line number in a program
. e.g. Delete 10 will delete line number 10
Delete 30-50 will delete all line numbers between 30 to 50.

THE GOTO STATEMENT

Quite often you don't want the program to run exactly in the order you put the lines, from the first to the last. Sometimes you want the program to jump to a particular line. You can use GOTO to jump both forward and backward to any line you want.

The syntax for GOTO statement is given below;

GOTO n where Nis the label.

A label is a line number. Generally, statements in QBASIC are assigned serial number for referencing. Always remember to include a label in a GOTO statement. You can have more than one label, but in that case they should be different.

READ AND DATA STATEMENTS

READ and DATA statements offer an alternative method of assigning values to variables. Below are two ways of assigning the variable X the value 2.

```
Method 1 :    X = 2          Method 2 :    READ X
                                           DATA 2
```

Here are four ways of assigning X the value 2 and Y the value 3:

```
X = 2          READ X, Y      READ X         READ X, Y
Y = 3          DATA 2, 3      READ Y         DATA 2
                              DATA 2, 3      DATA 3
```

When Basic comes to a READ statement containing a list of variables, it looks for a DATA statement containing a list of values. It reads the data values one at a time, in the order listed and beginning with the first DATA statement in the program. A DATA statement may appear anywhere in the program - Basic will find it. Some programmers place all DATA statements at the beginning of the program, some at the end, while others like to place a DATA statement right after the READ statement referring to it. All READ and DATA statements may refer to strings as well as numerical variables. Items in READ and DATA statements are separated by commas.

Example :

```
READ student$, class$, age, weight, homecity$
DATA Joe Smith, junior, 22, 154, "Honolulu, Hawaii"
```

In a DATA statement it is optional whether or not you put quotation marks around a string; however, if the string contains punctuation marks then you might need quotation marks to avoid ambiguity. If one typed just Honolulu, Hawaii in the above DATA statement, then Basic would interpret Honolulu and Hawaii as two different strings.

If a DATA statement is too long for the screen, divide it into two statements, as below:

```
DATA Ann Ames, Betsy Bates, Cindy Crim, Diana Dees, Eve Evans
DATA Freda Finley, Greta Garbo, Hilary Hill, Inez Ingall, Jan Joplin
```

Be certain that data types in your DATA statements match variable types in your READ statements. The sequence

READ A$, B
DATA horse, buggy

produces a "Syntax error" message, because "B" is a numeric variable but "buggy" is a string.

THE IF…THEN STATEMENT

This is a logical statement that is vital for altering the sequential flow of a program. The syntax for IF…THEN statement is given below;

IF (logical expressions) THEN stmt
where stmt represents a QBASIC statement.

Alternatively, we can express the IF…THEN statement as a block viz;

IF (logical expressions) THEN
Stmt(s)
ENDIF

Stmt(s) represent QBASIC statement(s)

If the logical expression or argument is true, the interpreter executes the statement(s) immediately after the THEN keyword. If the argument is not TRUE, then the QBasic bypasses this line(s) and goes to the next.

Example:**IF a = 15 THEN PRINT "OK"**

If the argument is not true (if a is not equal to 15), QBasic bypasses this line and goes to the next. In some cases, you can use the ELSE command, which tells QBasic exactly what to do if the argument is not true.

THE IF…THEN…ELSE STATEMENT

This statement has the following syntax;

IF (logical expressions) THEN stmtELSE stmt

Or

IF (logical expressions) THEN
Stmt(s)

ELSE
Stmt(s)
ENDIF

The ELSE command tells QBasic exactly what to do if the logical expression is not true.

You can make QBasic to execute more than one command if the argument is true. To do this, put those commands after IF…THEN and divide them with : symbol.

Example: IF a = 15 THEN PRINT "OK": GOTO 5
This example means that if a equals to 15, QBasic will first print OK and then will go to the line labelled 5

LOOPING STATEMENTS

When a set of instructions are repeatedly executed a fixed number of times it is termed as Loop. A loop simply means iteration or a repetition.

To make interesting and efficient programs, you can make QBasic to execute a part of a program more than once. This is called looping, when QBasic goes through a part of a program over and over again. This can be done with the GOTO command, but there are some good and more efficient ways to loop the program rather using GOTO statements. One of them is FOR...NEXT command.
Loops can be implemented using counters to keep track of the number of repetitions to done. When using Counters, we have to follow the following points:

1. Initialize the counter
2. Increment or Decrement the counter
3. Check for the maximum limit

Example of a counter includes; suppose the variable I is initialize to zero, a counter will be implemented as follows:

$$i = i + 1$$

The term Counter is used to describe a variable that we use to count up;

- how many things that we have
- how many times something is repeated

NOTE: counters are most useful in loops and if statements

FOR…NEXT LOOP

This command allows you to execute a part of a program a certain number of times. It has the syntax as shown below:

FOR var = x TO y [STEP z]

Program statements…

NEXT [var]

Where

- var is a variable. It is a counter and is called the 'Control Variable'.
- x is the start value
- y is the final or ending value

- x and y are numeric values
- var is assigned all values starting with x terminating with y in steps of z

NB: The [step] size at the loop header is optional. If STEP is not given then BASIC assumes the increment to be 1. Also, var increments or decrements depending on whether the number specified in STEP is positive or negative.
Example include;

```
FOR a = 1 TO 5
PRINT "This is loop number"; a
NEXT a
```

This will print:

```
This is loop number 1
This is loop number 2
This is loop number 3
This is loop number 4
This is loop number 5
```

The STEP command can be implored as shown below. This will tell QBasic how to count from one number to another:

```
FOR j = 0 TO 12 STEP 2
    statements….
NEXT j
```

The above loop will count in two's: 0, 2, 4, 6, 8, 10, 12.
Examples include:
1.Program to print numbers 10 to 15 using For…Next loop

```
10    Let N = 15
20    FOR M = 10 TO N
30    Print M;
40    Next M
```

Output as displayed on screen:

10 11 12 13 14 15

2. Program to print numbers 40 to 50 in reverse order using For…Next loop

```
10    FOR A = 50 TO 40 STEP –1
20    Print A;
30    Next A
```

Output as displayed on the screen:50 49 48 47 46 45 44 43 42 41 40

3.Program to print first 10 multiple of any number input.

```
10 Input “Enter any number “,n

20 for I= 1 to 10

30 Print n; ”x” ;I “=” n* i

40 next

5

0

e

n

d
```

Output :

```
10      x    1       = 10

10      x    2       = 20

10      x    3       = 30

10      x    4       = 40

…….
```

WHILE...WEND

The WHILE...WEND commands continue a loop until a specified expression is false.

To use WHILE...WEND:

1. Place an expression after WHILE
2. Enter a list of commands
3. Place WEND at the end

The general syntax for the WHILE…WEND loop is as follow;

WHILE [logical expression]

    Stmt-1
    Stmt-2
    …
    Stmt-n

WEND

Example include;

```
5                                                    x = 10
10  WHILE x < 15
15                                                   PRINT x
20      x = x + 1
25  WEND
```

Recall: x=x+1 is a counter

Output :

10
11
12
13
14

THE DO… LOOP

This loop uses both WHILE
and UNTIL keywords.
WithDO...LOOP you can:

1. Loop until an expression is **true**
2. Loop at least **one** time regardless of whether the expression is true or not.

To use DO...LOOP:

1. Specify whether the loop continues **"while"** the expression is true or**"until"** the expression is true, using the WHILE and UNTIL statements, respectively.
2. Place an expression after WHILE/UNTIL
3. Enter a list of commands
4. Place LOOP at the end

The syntax for the Do…WHILE loop is given below;

```
DO WHILE [logical expression]
    Stmt-1
    Stmt-2
    …
    Stmt-n
LOOP
```

The following uses the WHILE statement:

```
x = 10
DO WHILE x < 15
        PRINT x
        x = x + 1
LOOP
```

The syntax for DO…UNTIL loop is given below;

```
        DO UNTIL [logical expression]
                Stmt-1
                Stmt-2
                …
                Stmt-n
        LOOP
```

This program uses the UNTIL statement:

```
x = 10
DO UNTIL x = 15
        PRINT x
        x = x + 1
LOOP
```

The both output:

```
10
11
12
13
14
```

If you place the expression at the end of the loop instead, the program goes through the loop at least once.

```
x = 32
DO
        PRINT x
        x = x + 1
LOOP WHILE x < 5
```

This is the output because the loop was only gone through one time:

```
32
```

The Sample program for a DO WHILE Loop

```
*** This program adds a list of positive integers. ***
CLS
Sum = 0
INPUT "Enter the first number (-l to quit): ", Number
DO WHILE Number <> -1
Sum = Sum + Number
INPUT "Enter the next number (quit if -1): ", Number
```

```
LOOP
PRINT "The sum is"; Sum
END
```

Program Output

```
Enter the first number (quit if -1): 24

Enter the next number (quit if -1): 18

Enter the next number (quit if -1): 91

Enter the next number (quit if -1): -1

The sum is 133
```

Notice that Sum is set to zero immediately before the loop is entered. Technically, this statement is not needed. QBasic automatically sets the values of numeric variables to zero and string variables to the null (empty) string before execution begins. However, it is good programming practice to initialize variables yourself, rather than depending on "default" initializations.

Nested loops:
You can put a loop inside of another loop in a structure called nested loops. This is done by placing one **For...Next loop** within another. Each loop **MUST** have a unique variable name as its loop counter.
Nested loop example:

```
For I = 1 To 10
        For J = 1 To 10
                For K = 1 To 10
                        Statement(s)
                Next K
        Next J
Next I
```

NB: Remember that nested loops must be bracketed in LIFO order. That means that the inner-most loop ends first and the outermost loop ends last.

The EXIT Statement
Occasionally, it may be necessary to exit a loop prematurely. Most commonly, this occurs if an error condition is encountered; for example, if invalid data has been read. The EXIT statement can be placed anywhere in a DO... LOOP and causes execution to be immediately transferred to the first statement after LOOP For

example, the following program segment is supposed to read the names of 12 voters. However, if a value of less than 18 is read for Age, an error message is displayed and the loop terminates.

```
Rem Read and display names of 12 voters. ***
DO WHILE Count <= 12
READ Voter, Age
IF Age < 18 THEN
PRINT
PRINT Voter ; " is not eligible to vote."
EXIT DO
END IF
PRINT Voter$
Count = Count + 1
READ Voter$, Age
LOOP
Rem Data statements
DATA Santana, 18, Matthews, 44, Ericson, 17, Ling, 29
END
```

If the EXIT DO statement is executed, control transfers to the first statement after LOOP However, this statement is executed only if an invalid value for Age is encountered. The EXIT statement can he used with other structures, such as SUB procedures. The statement EXIT SUB allows control to be transferred immediately back to the calling program. However, a word of warning: The EXIT statement can lead to program errors. It also makes program logic more difficult to follow. Therefore, it is best to avoid its use unless absolutely necessary.

Counters vs Accumulators

The term **Accumulator** is used to describe a variable that is being used to total up a bunch of numbers.To use a variable as an accumulator, you must follow these two steps:

1. Set the variable that will be used as an accumulator to a starting value (usually you will start theaccumulator at '0' [zero])
2. Next add something to what is already stored in the accumulator

(for example: Accumulator = Accumulator + someNumber)

NOTE: Accumulators are often found in loops because they repeat the process of "accumulating" numbers.

In the QBASIC code shown below, the variable named "TOTAL" is used as an accumulator.

```
LET TOTAL = 0.0
LET AVG = 0.0
CLS
PRINT "Please enter five marks"
FOR I = 1 to 5
INPUT "Enter mark: "; mark
TOTAL = TOTAL + mark
```

```
NEXT I
AVG = TOTAL / 5
PRINT "The total of the marks is ";TOTAL
PRINT "The average of the marks is ";AVG
END
```

ARRAYS

An array is a list of variables of the same type. Arrays are useful for organizing multiple variables. To create an array, use the **DIM** (dimension) command.
The general syntax for declaring array is: DIM varname (arraysize),
Where varname is the array name; arraysize is the size of the array

The following example does *not* use arrays:

```
a = 2
b = 4
c = 6
d = 8
e = 10

PRINT a, b, c, d, e
```

Output:

```
2       4       6       8       10
```

This uses an array called **vars**, which contains 5 variables:

```
DIM vars(5)

' Each of these are separate variables:
vars(1) = 2
vars(2) = 4
vars(3) = 6
vars(4) = 8
vars(5) = 10

PRINT vars(1), vars(2), vars(3), vars(4), vars(5)
```

Output:

```
2       4       6       8       10
```

The above program can also be written like this:

```
DIM vars(5)

FOR x = 1 to 5
        vars(x) = x * 2
NEXT

FOR x = 1 to 5
        PRINT vars(x),
```

```
NEXT
```

Output:

2 4 6 8 10

QBASIC MATHEMATICAL LIBRARY FUNCTIONS:

- ABS : Full form is Absolute value. It returns the absolute value of a number.
  Syntax : <Numeric Variable> = ABS <Numeric Variable>
  Example

```
10          cls

20          let A = -12

30          B = ABS(A)

40          print b

50          end
```

  The output of the above program would be 12.

- SQR: Full form is Square Root. It calculates the square root of the value represented by the variable.
  Syntax : <Numeric Variable> = SQR <Numeric Variable>
  Example

```
10          cls

20          let a = 16

30          b = SQR(a)

40          print b

50          end
```

  The output of the above program would be 4.
  NOTE : The computer will only give square root of positive number else it will give the error : ILLEGAL FUNCTION CALL.
- INT: Full form is Integer. It returns only the integer portion of a real number. The real number can be positive or negative or decimal number. INT only accepts the real number but ignores its the decimal part.
  Syntax : <Numeric Variable> = INT <Numeric Variable>
  Example;

```
10  cls

20      let a = 13.5

30      let b = -90.9

40      c = INT(a)
50      d = INT(b)
```

```
60      print c, d

70      end
```

The output of the above program would be 13 and 90 respectively.

- MOD : It returns the integer remainder .
  Syntax :

  <Numeric variable>= <integer variable/constant> MOD <integer variable/constant>

  e.g.

```
10 cls

20 Let a=13

30 c= a MOD 2

40 print c

50 end
```

Output is :1(for 13 divided by 2 gives an integer remainder 1)

TEXT LIBRARY FUNCTIONS

LEN : Returns the length of the string. Syntax :

```
L = Len (string);
```

Where, L = the variable storing the no of characters the string has.
String = the string whose length has to be found.
Example:

```
10      cls

20      s$ = "Tinapa"
30      m = len(s$)
40      print m
50      end
```

The above program would give output = 6, as even space is considered as a character.

ASSIGNMENT: Study additional functions that apply to QBASIC programming language.

PRACTICAL WORK

ACTIVITY ONE: Write a program to compute n factorial where n=10 and is inputted from the console.

ACTIVITY TWO: Write a program to sum odd numbers between 1-100

ACTIVITY THREE:

Write a program to create a 1-dimensional array containing 5 elements. Compute the sum, average and standard deviation of the array elements

ACTIVITY FOUR:

Write a program to read values for the integer variables a, b, and c and compute the roots of the equation: $ax^2 + bx + c = 0$ using the formula $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$